VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Alena Martinková

Bakalářská práce

Vedoucí práce: Ing. Ján Kožusznik Ph.D.

Ostrava, 2021

Abstrakt

Táto bakalárska práca popisuje priebeh absolvovania individuálnej odbornej praxe vo firme Moravio s. r. o. Prvá časť sa zaoberá firmou samotnou, jej technickým zameraním a mojou pracovnou náplňou. Druhá časť popisuje projekt, na ktorom som pracovala, úlohy, ktoré mi boli pridelené, ich zadanie, popis ich riešenia a výsledky. V poslednej kapitole zhodnotím zručnosti získané vykonaním praxe.

Kľúčové slová

odborná prax, Moravio s. r. o., PHP, Laravel, JavaScript, MySQL, HTML

Abstract

This bachelor thesis describes the course of completing individual professional practice in the company Moravio s. r. o. The first part deals with the company itself, its technical focus and my job. The second part describes the project I have been working on, tasks that were assigned to me, their assignment, a description of their solution and results. In the last chapter I will evaluate the skills acquired through practice.

Keywords

professional practice, Moravio s. r. o., PHP, Laravel, JavaScript, MySQL, HTML

Poďakovanie

Na tomto mieste by som rada poďakovala firme Moravio s. r. o. za možnosť absolvovania bakalárskej praxe a môjmu vedúcemu práce Ing. Jánovi Kožusznikovi Ph.D. za poskytnutie rád a informácií, ktoré umožnili zdarné ukončenie tejto práce.

Obsah

Zoznam použitých skratiek a symbolov

API	–	Application Programming Interface
CSS	–	Cascading Style Sheet
SASS	–	Syntactically Awesome Style Sheet
SCSS	–	Sassy CSS
HTML	–	Hypertext Markup Language
JS	–	JavaScript
PHP	–	JavaScript
SQL	–	Structured Query Language
MVC	–	Model-View-Controller
ORM	–	Object-Relation Mapping
GUI	–	Graphical User Interface

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Ako tému svojej bakalárskej práce som si zvolila absolvovanie bakalárskej praxe vo firme Moravio s. r. o. (ďalej len Moravio). Dôvodom voľby praxe bolo predovšetkým získanie skúseností do ďalšieho profesijného života, no taktiež okúsenie firemnej kultúry, či otestovanie svojich vedomostí dosiahnutých počas doterajšieho štúdia na vysokej škole. V predchádzajúcom semestri som taktiež absolvovala predmet Prax, práve vo firme Moravio. Táto skutočnosť ma povzbudila pri voľbe mojej bakalárskej práce.

Bakalárska práca je rozdelená do troch rozsiahlejších častí. V prvej z nich sa budem venovať firme Moravio, popisu môjho pracovného zaradenia a použitých technológií počas bakalárskej praxe. V druhej časti popíšem projekt, na ktorom som pracovala, vybrané úlohy týkajúce sa vytvorenia nových funkcionalít, či úpravy už existujúcich funkcionalít v danom projekte, zvolený postup riešenia a výsledky. Posledná časť práce bude obsahovať zhodnotenie uplatnenia štúdiom dosiahnutých vedomostí a taktiež získaných teoretických i praktických skúseností počas vykonávania odbornej praxe.

Kapitola 2

Popis firmy a jej odborného zamerania

2.1 Firma Moravio

Firma Moravio bola založená v roku 2011, takže je možné ju považovať za pomerne novú, mladú firmu. Spoločnosť má svoje kancelárie v Ostrave. V dôsledku koronavírovej situácie väčšina vývojárov pracuje na diaľku, kolegovia vykonávajú činnosť po celej Českej republike. Prácu poskytuje viac než 20 zamestnancom, resp. spolupracovníkom. V súčasnosti je hlavným cieľom firmy rozšíriť pôsobenie do viacerých krajín.

2.2 Odborné zameranie

Zameraním firmy Moravio je návrh, vývoj a správa webových či mobilných aplikácií. Projekty tak vedie od úplného začiatku – návrh systému spolu s jeho grafickým designom, samotné programovanie, spustenie a následná správa, poprípade ďalší rozvoj. Firma sa skladá z viacerých menších vývojárskych tímov, ktoré pracujú na svojich projektoch, produktového, projektového, administratívneho a marketingového tímu. Každý vývojársky tím má projekt, na ktorom pracuje, pričom niektorí zamestnanci, hlavne juniori, sú vyčlenení taktiež do tzv. Supportu, ktorý má za úlohu starostlivosť o zákazníkov a riešenie akútnych problémov na weboch. Programuje sa hlavne v PHP, spolu s JavaScriptom, za využitia rôznych frameworkov – najčastejšie Wordpress alebo Laravel. Postupne sa však posúva spolu s novými technológiami a začína sa pracovať v React.js, Node.js či Vue.js. Moravio má za sebou množstvo veľkých projektov, či už ich správu alebo celý vývoj, napr.:

- Mall.cz
- DameJidlo.cz s.r.o.
- Kofola ČeskoSlovensko a.s.
- Notino, s.r.o.

- CENTRL, Inc.

Za zmienku určite stojí fakt, že v roku 2019 Moravio nadviazalo aj spoluprácu s americkou firmou JLL, s ktorou neustále vyvíja virtuálnu hlasovú asistentku JET. Po dobrých skúsenostiach sa spolupráca s americkou firmou predĺžila a rozšírila. Okrem vývoja spomínaných aplikácií a systémov je Moravio taktiež spoluorganizátorom konferencie Barcamp, ktorý sa v minulosti konal práve na Vysoké škole báňské.

2.3 Pracovné zaradenie

V rámci vykonávania bakalárskej praxe som pracovala spolu s dvoma kolegami, taktiež študentmi VŠB, na PHP projekte na pozícii programátorky/kóderky. Mojou úlohou bolo nielen samotné programovanie zadaných úloh, ale takisto aj navrhovanie nových riešení pre dané problémy. Počas vývoja projektu sme s kolegami a vedúcim projektovým manažérom viedli týždenné statusy, v ktorých sme ostatných informovali, ako postupujeme v daných úlohách. Na rozdelenie, časové odhadnutie úloh a prípadné spätné dohľadanie úloh bola používaná aplikácia Jira. Čas strávený na daných úlohách bol monitorovaný pomocou aplikácie Toggl. Projekt samotný opíšem detailnejšie v štvrtej kapitole.

Kapitola 3

Použité technológie

V tejto kapitole popíšem technológie, ktoré sú použité v projekte a taktiež v úlohách mnou vypracovaných.

3.1 PHP

PHP je open source skriptovací jazyk, ktorý sa používa najmä na programovanie klient-server aplikácií a pre vývoj dynamických webových stránok. Celý kód je vykonávaný pomocou PHP runtime, pre dynamické vytváranie obsahu na webovej stránke. PHP je schopné spracovať kód ohraničený špeciálnymi tagmi. Najčastejšie používané sú `<?php` a `?>`. Použiť sa dá aj `<script language="php">` a `</script>` [1]. V Laravel šablónach je taktiež možné využiť špeciálnu syntax a to `@php` a `@endphp`. Existujú v ňom tri druhy komentárov:

- `/* ... */` - blokový komentár,
- `// ...` – jednoriadkový komentár,
- `# ...` – jednoriadkový komentár.

Premenné sa označujú symbolom `$` a je možné ich pretypovať.

3.2 Laravel

Laravel je PHP framework používaný na vývoj webových aplikácií. Je založený na architektúre MVC. Model obsahuje aplikačné dáta, ak je použité ORM, korešponduje priamo s databázovými tabuľkami. View slúži ako prezentačná vrstva, väčšinou písaná v HTML. Controller spravuje interakciu medzi užívateľom, Modelom a View[2]. Životný cyklus stránky vyzerá nasledovne:

- užívateľ zadá do prehliadača adresu,

- túto adresu zachytí router, ktorý podľa parametrov spozná, ktorý Controller voláme,
- Controller vykoná potrebnú logiku, zavolá Model a vráti jeho údaje,
- tieto údaje vygeneruje View – reprezentuje ho šablóna s názvom „názov.blade.php“.

3.3 HTML

HTML je značkovací jazyk určený na vytváranie webových stránok a ďalších informácií zobraziteľných vo webovom prehliadači[3]. Základom HTML sú tagy a atribúty. Tagy sú používané na označenie začiatku, resp. konca elementu. Rozdelené sú na párové a nepárové. Atribúty obsahujú pridané informácie o danom elemente, poprípade dáta, ktoré potrebujeme použiť napr. v JS. Príkladom atribútu je aj *src* či *alt* v elemente *<img>*.

Každý HTML dokument je rozdelený na hlavičku a telo, ktoré sú vnorené do koreňového elementu *<html>*. Elementy môžu obsahovať triedy a identifikátory pre možnosť lepšieho zacielenia daného elementu pomocou CSS, či JS.

3.4 CSS/SASS/SCSS

CSS je tzv. jazyk šablón štýlov, ktorý slúži na úpravu vzhľadu elementov HTML. Možnosť výberu elementov na základe spoločnej triedy, či ID umožňuje ich rovnaké naštýlovanie jedným riadkom kódu. Kód je uložený v súboroch s príponou .css.

SASS je preprocesorový skriptovací jazyk, ktorý je rozšírený o množstvo užitočných vlastností a následne skompilovaný do CSS. Dáva možnosť využívať premenné, zanorovať, používať funkcie, argumenty a ďalšie.

SCSS je novšia syntax SASS. Využíva príponu .scss.

3.5 JavaScript/jQuery

JavaScript je skriptovací programovací jazyk. Popri CSS a HTML je jednou z hlavných technológií používaných pri programovaní webových stránok. Používa sa vo veľkej miere na riešenie dynamiky na strane klienta, avšak je možné na ňom postaviť aj celú frontendovú aplikáciu[4]. Je multiplatformný, beztypový, závislý na interpretačnom prostredí ai.

V projekte je často využívaná knižnica jQuery, ktorá uľahčuje prácu v JS. Všetok kód napísaný v jQuery sa skonvertuje do JS, rozdiel je však v počte riadkov kódu, ktorý je potrebný na vykonanie rovnakej akcie. V kóde sa jQuery používa väčšinou pomocou znaku $.

3.6 Bootstrap

Bootstrap je bezplatná a open source JS a CSS knižnica, ktorá je zameraná na responzívny vývoj aplikácií. Obsahuje šablóny pre typografiu, formuláre, tlačidlá, navigáciu ai.

Najpoužívanejšou časťou Bootstrapu počas tejto bakalárskej praxe bolo rozdelenie obsahu na stránke do tzv. riadkov a stĺpcov. Jeden riadok vždy obsahuje 12 stĺpcov. Následne pomocou bodov zlomu daných touto knižnicou umožňuje jednoduchú a rýchlu úpravu desktopovej aplikácie na responzívnu aplikáciu pre mobilné zariadenie. Body zlomu sú: 768px(sm), 992px(md), 1200px(xl) a 1400px(xxl). Podľa týchto bodov zlomu sú pomenované aj príslušné triedy, ktoré sa používaju v HTML. Napr. element s triedami *col-md-6* a *col-sm-12* bude nad 992px vrátane, zobrazený na polovicu z daného riadku a pod 992px na celú šírku daného riadku.

3.7 Git

Git je bezplatný a open source systém slúžiaci na kontrolu a riadenie projektov – od malých po veľké. Vo firme Moravio sa Git používa na všetky projekty, takže v rámci praxe sme sa naučili pracovať s ním pomerne rýchlo. Na jednoduchšie ovládanie sa používa software GitKraken, ktorý je intuitívnym GUI nástrojom pre prácu s Gitom.

V Gite sa využívajú na prácu vetvy, ktoré slúžia na to, aby každý programátor mohol vykonávať svoju časť práce bez toho, aby prepisoval úpravy iných programátorov. V prípade, že dvaja programátori upravia tú istú časť kódu, nastane tzv. konflikt, na ktorý Git upozorní a umožní vyriešiť ho. Hlavné vetvy sú „master“, „devel“ a „feature“.

3.8 Docker

Docker je open-source softvér, ktorý umožňuje rozdelenie aplikácii do izolovaných prostredí, tzv. kontajnerov. Každý kontajner má svoje vlastné nastavenia a konfiguračné súbory. Kontajnery medzi sebou komunikujú cez kanály[5]. Na spúšťanie, resp. vypínanie docker kontajnerov cez terminál sa používajú príkazy `docker-compose up` a `docker-compose down`, avšak je možné používať aj GUI.

Hlavnou myšlienkou a výhodou Dockeru je fakt, že všetky nastavenia kontajnerov a teda aj celého projektu sú v tzv. docker images, čo umožňuje spustiť projekt na akomkoľvek zariadení s totožnou konfiguráciou.

Kapitola 4

Projekt Spanamo

Jedná sa o PHP projekt, postavený na frameworku Laravel, výsledkom ktorého bude medzinárodný portál zameraný na rezervácie kúpeľných hotelov, liečebných pobytov a procedúr. Bol prevzatý ako rozpracovaný projekt, ktorý vyvíjali zahraniční programátori, no nebol dotiahnutý do konca. Kód bol v zlom stave a nedodržiaval základné princípy MVC vzoru, na ktorom je Laravel založený.

Dôležitým faktorom v projekte sú typy užívateľov. Existujú štyri, hierarchicky - super-admin, admin, hotel manažér a klient. Každý typ užívateľa má iné povolenia v rámci projektu. Povolenia sa taktiež dajú explicitne nastaviť pre každého užívateľa samostatne.

Projekt je rozdelený na dve webové stránky – každá je samostatný projekt, vedený vo svojom vlastnom Git repozitári. Tieto projekty zdieľajú jednu MySQL databázu.

Prvý je frontendová stránka, na ktorú budú mať prístup užívatelia, v terminológii projektu klienti, ktorí majú záujem o rezerváciu hotela s rôznymi procedúrami. V prípade potreby sa do frontendovej stránky môže prihlásiť aj super-admin. Žiadny iný typ užívateľa do frontendovej časti prístup nemá. Užívateľ sa môže prihlásiť do svojho účtu, kde vidí svoje osobné informácie, doterajšie objednávky, obľúbené hotely a iné. Portál je viacjazyčný a pre preklady všetkých textov je vytvorený vlastný modul. Ďalej v bakalárskej práci budem na túto časť projektu odkazovať ako na **Front**.

Druhý je tzv. systém, do ktorého majú prístup super-admini, admini a hotel manažéri. Tu sa vytvárajú a upravujú hotely, izby, procedúry, ceny, už spomínané preklady a vykonávajú všetky potrebné administratívne činnosti. Ďalej v bakalárskej práci budem na túto časť projektu odkazovať ako na **Systém**.

Projekt je postavený na Dockeri a rozdelený do viacerých Docker kontajnerov. Každý z nich sa stará o chod samostatnej časti projektu. Počet kontajnerov na testovacej, resp. produkčnej časti je odlišný, ako na lokálnej.

- **app-front** - postavený na PHP 7.2.-fpm, obsahuje odkazy na všetky súbory Frontu. V lokálnej časti na súbory priamo v počítači, v testovacej, resp. produkčnej časti je uložený v Gitlab registroch a odkazuje na dynamické časti Frontu.

- **app-system** - postavený na PHP 7.2.-fpm, obsahuje odkazy na všetky súbory Systému. V lokálnej časti na súbory priamo v počítači, v testovacej, resp. produkčnej časti je uložený v Gitlab registroch a odkazuje na dynamické časti Systému.
- **webserver** - obsahuje Nginx server. V testovacej, resp. produkčnej časti slúži ako koreňový server, ktorý presmerováva na webserver-front, resp. webserver-system.
- **webserver-front** - obsahuje Nginx server spolu so statickými súbormi Frontu, napr. obrázky. Tento kontajner nie je v lokálnej časti.
- **webserver-system** - obsahuje Nginx server spolu so statickými súbormi Systému, napr. obrázky. Tento kontajner nie je v lokálnej časti.
- **mysql** - obsahuje MySQL databázu.
- **adminer** - beží na porte 8001 a umožňuje prístup do databázy pomocou GUI.
- **sphinx** - obsahuje Sphinx fulltextový vyhľadávací engine.

Ako som spomínala, používali sme Git - náš workflow obsahoval základné vetvy ako „master“, „devel“, „feature“ a pre týždenné šprinty vždy jeden „release“, do ktorého boli zlúčené všetky „feature“ z daného šprintu. Vzhľadom na to, že ide o projekt, ktorý je stále vo vývoji, nebolo nutné používať vetvu „hotfix“.

V projekte som pracovala ako kóderka/programátorka. Mojou úlohou bolo nielen programovať zadané úlohy, no taktiež aj navrhovať efektívne a vhodné riešenia, ktoré boli následne prekonzultované v rámci nášho tímu. Úlohy obnášali reštrukturalizácia kódu, úpravy existujúcich modelov a následne vytváranie nových modelov, ktoré v projekte ešte neboli, poprípade boli, no ich funkčnosť bola takmer nulová. V ďalších podkapitolách popíšem úlohy mne zadané a ich následné riešenie.

Úlohy boli zadávané v týždňových šprintoch, ktoré boli plánované vždy v piatok týždeň vopred. Projekt bol vedený agilne, klient sa teda aktívne podieľal na vývoji projektu a pravidelne nám poskytoval užitočnú spätnú väzbu. V prípade potreby konzultácie nám taktiež bola umožnená priama komunikácia s klientom, ktorý nám danú úlohu dodatočne vysvetlil. Každý piatok sme taktiež zhodnocovali dokončený šprint a referovali prípadné problémy, na ktoré sme narazili.

4.1 Filter/konfigurátor obsadenosti izieb hotelov

4.1.1 Zadanie úlohy

Mojou prvou úlohou bola úprava už existujúceho filtra, resp. konfigurátora obsadenosti hotelov v Systéme. Pôvodne bolo možné filtrovať výpis hotelov len na základe jeho názvu, dátumu a dní v týždni. Taktiež bol rozdiel v tom, čo ktorý užívateľ vidí. Hotel manažérovi boli zobrazené len jeho hotely, naopak super-admin a admin ich mohli vidieť všetky. Hotel manažér mal navyše v bočnej lište

povolené zmeniť pre jeden hotel v istom časovom rozmedzí obsadenosť, poprípade status zvolených izieb. Filter aj konfigurátor sa pôvodne nachádzali pod jednou URL, v rámci zadanej úlohy bol konfigurátor oddelený od filtra. Zadanie obsahovalo dve väčšie časti.

Prvou z nich bolo pridanie piatich nových filtrov a úprava samotného výpisu výsledku na základe grafického návrhu zadaného klientom. Každý z týchto filtrov na seba nadväzoval a teda výsledky v každom filtri boli adekvátne tomu, čo sa nachádza vo filtroch predchádzajúcich.

Druhá časť obsahovala oddelenie konfigurátora od filtrov samotných. Pre konfigurátor bola vytvorená nová URL, ktorá umožňovala zobraziť priamo izby zo zvoleného hotela a úpravu len jeho samostatne. Rozhodnutie oddeliť tieto dve časti padlo z veľkej časti na základe zrýchlenia stránky a zväčšenia prehľadnosti. Takisto bolo logické, pretože nakonfigurovať rovnako dostupnosť, resp. nedostupnosť izieb je špecifické pre každý jeden hotel a v tom prípade je nepotrebné až nadbytočné zobrazovať aj iné hotely. Znamenalo to taktiež čistejší a prehľadnejší grafický návrh.

4.1.2 Návrh riešenia a jeho implementácia

Vzhľadom na stav kódu od predchádzajúcich vývojárov som sa rozhodla pre jeho kompletné prepísanie. Bolo to efektívnejšie ako upravovať kód, ktorý z veľkej časti nefungoval požadovaným spôsobom.

V prvom rade som vytvorila nový Controller, ktorý spracováva potrebné požiadavky a taktiež nové modely, ktoré Controller využíva. Z databázy som zmazala nepotrebné údaje.

Následne som naimplementovala nové filtre. Boli to filtre podľa: krajiny, lokality, hotela, hviezdičiek hotela, kategórie izby a samotnej izby, dňov v týždni a dátumu. Fungovali ako *<select>* elementy, ktoré obsahovali *<options>*. Hodnotu daných *<options>* som nastavila tak, aby sa v následnom volaní ajax požiadavky do databázy dalo vyhľadávať čo najjednoduchšie, bez zbytočných dotazov. Vzhľadom na množstvo dát, v ktorých algoritmus musí vyhľadávať bolo množstvo dotazov do databázy hlavným parametrom, ktorý ovplyvňoval efektivitu. V nasledujúcom texte v krátkosti popíšem každý z filtrov.

Filter krajiny: obsahuje všetky aktívne krajiny z databázy. Užívateľ vždy môže vybrať viacero krajín zároveň. Výpis v *<options>* zobrazuje vždy { ID } | { názov krajiny v zvolenom jazyku }. Hodnota je ID krajiny v databáze.

Filter lokality: obsahuje všetky aktívne lokality zo zvolených krajín z predchádzajúceho filtra. V prípade, že nie je zvolená žiadna krajina, sú zobrazené všetky aktívne lokality. Výpis v *<options>* zobrazuje vždy { ID } | { názov lokality v zvolenom jazyku }. Hodnota je ID lokality v databáze.

Filter hotelov: obsahuje všetky aktívne hotely zo zvolených lokalít z predchádzajúceho filtra. V prípade, že nie je zvolená lokalita, obsahuje všetky aktívne hotely zo zvolenej krajiny. Ak nie je zvolená ani krajina, obsahuje úplne všetky aktívne hotely. Výpis v *<options>* zobrazuje vždy { ID } | { názov hotela v zvolenom jazyku }. Hodnota je ID hotela v databáze.

Filter hviezdičiek: obsahuje možnosti počtu hviezdičiek, ktoré majú aktívne hotely. Teda ak sú aktívne len päťhviezdičkové hotely, v *<options>* sa zobrazí len táto možnosť. Tak isto ako

predchádzajúce filtre nadväzuje na možnosti zvolené v nich, ak zvolíme lokalitu, ktorá má len dvoj- a trojhviezdičkové hotely, zobrazí sa len táto možnosť. Hodnota je ID danej možnosti v databáze.

Filter kategórie izby: obsahuje na výber kategórie izieb, ktoré sa nachádzajú v danej lokalite, resp. hoteli. Kategóriou rozumieme jedno-, dvoj- či trojposteľová izba, popr. apartmán. Pokiaľ nie je zvolená minimálne lokalita, obsahuje len možnosť All. Výpis v <options> zobrazuje vždy { ID } | { názov typu izby v zvolenom jazyku }. Hodnota je ID daného typu izby v databáze.

Filter izby: obsahuje samotné izby hotela. Pokiaľ nie je zvolený konkrétny hotel, obsahuje len hodnotu All. Výpis v *<options>* zobrazuje vždy { názov izby v zvolenom jazyku } ({ počet postelí + prísteliek v danej izbe }). Hodnota je ID danej izby v databáze.

Filter dní: obsahuje dni v týždni. Výpis v *<options>* zobrazuje vždy { názov dňa v zvolenom jazyku }. Hodnota je ID daného dňa v databáze.

Filter dátumu: obsahuje výber dátumu od – do za použitia externej knižnice. Za zvolené obdobie sa zobrazí výsledok filtrovania.

V každom filtri pre preklady platí nasledujúce: v prípade, že preklad v danom jazyku neexistuje, zobrazí predvolene preklad v anglickom jazyku.

Keďže filtre musia fungovať dynamicky, je nutné, aby sa vždy pri zmene hodnoty v danom filtri taktiež zmenili hodnoty v ďalších nadväzujúcich filtroch. Z tohto dôvodu je nutné odchytávať zmeny v *<select>* elementoch a následne volať ajax požiadavku, ktorá vyfiltruje nové korektné dáta. Na zobrazenie *<select>* som použila knižnicu select2 a v tomto prípade sa na odchytávanie týchto zmien nepoužíva JS funkcia `onchange`, ale select2 funkcie, ktoré ukážem v nasledujúcom výpise kódu.

```
// on select2 selecting
$(".select2")
  .select2({ selectOnClose: false })
  .on("select2:selecting", function (evt) {
    // if all is selected unselect everything else
    if (evt.params.args.data.id == 0) {
      $(evt.target).val(null).trigger("change");
    } else {
      // if some other value is selected unselect all option
      $(".select2 > option[value=0]").prop("selected", false);
    }
  });

// on select2 unselect/select
$(".select2-filter").on("select2:select select2:unselect", function () {
  let selectValues = {};
  // get all values
  $("select.select2-filter").each(function (index, element) {
```

```
    selectValues[$(element).attr("name")] = $(element).val();
  });

  // get hotel id
  let id = $("#hotel-id-input").val();

  // call ajax for filtering
  $.ajax({
    type: "POST",
    url: window.location.origin + "/allotment-filter",
    data: {
      // data sent to ajax
    },
    success: function (data) {
      // fill all inputs
    },
    error: function (data) {},
  });
});
```

Listing 4.1: Ukážka kódu použitia select2

Týmto JS kódom sa vykoná ajax, ktorý zavolá funkciu z Controllera. V tejto funkcii algoritmus vyfiltruje dáta, ktoré sa majú zobraziť v ostatných *<select>* elementoch. Ďalej popíšem spomínaný algoritmus z Controllera.

V prvom rade sa skontroluje, či užívateľ, ktorý je prihlásený, má práva na to, aby videl dané dáta. Deje sa to na každej stránke, v každej funkcii, ktorá pracuje s dátami, ktorých zobrazenie má obmedzenia na základe užívateľských rolí. V tomto volaní, ktoré používa POST metódu, to musí byť taktiež, pre prípad, že by sa užívateľ snažil pristúpiť napr. pomocou URL. V prípade, že užívateľ nemá dané práva, funkcia sa ukončí a presmeruje uživateľa na URL /unauthorized. Ak práva má, funkcia pokračuje ďalej. Následne sú na základe typu užívateľa vyberaté hotely a lokality, ktoré sa budú filtrovať. Ak je užívateľ super-admin, resp. admin, je výber jednoduchý - všetky hotely a všetky lokality. V prípade, že je užívateľ hotel manažér, uvidí len hotely, ktoré má priradené a lokality, v ktorých sa hotely nachádzajú. Ak je užívateľom zvolená minimálne jedna krajina, vyfiltrujú sa len lokality s touto krajinou, resp. krajinami. To isté platí v prípade, že je zvolená minimálne jedna lokalita - vyfiltrujú sa len hotely v danej lokalite, resp. lokalitách. Avšak v pomocnej premennej sú uložené všetky lokality, ktoré sú v danej krajine - stále sa musia zobraziť v *<select>* elemente, len nebudú označené ako *selected*. V prípade, že je užívateľom zvolený minimálne jeden hotel, všetky nasledujúce filtre - hviezdičky, kategórie izieb a izby, filtrujú z jeho, resp. ich informácií.

Vyfiltrované lokality sú prejdené v cykle a sú zapojené zostávajúce filtre. To znamená, že sú vyfiltrované hotely, ktoré spĺňajú zvolené hodnoty vo filtroch hviezdičiek a kategóriách izieb. Vzhľadom na to ako algoritmus funguje, sa nikdy nemôže stať, že vo filtri nezostane ani jeden hotel a to z jednoduchého dôvodu - užívateľovi sú vždy navrhnuté len tie možnosti, pre ktoré existuje aspoň jeden korektný hotel. V prípade, že užívateľ najprv vyberie hotel, v nasledujúcich filtroch už vidí len hodnoty prislúchajúce hotelu. Naopak, ak najprv zvolí napr. hviezdičky, už si môže vybrať len z hotelov s daným počtom hviezdičiek. Dôležité v tomto algoritme je aj poradie kontroly, vždy je nutné vyfiltrovať najprv správne kategórie izieb a až následne z nich filtrovať izby samotné. Posledným krokom je návrat HTML, ktoré je dynamicky dosadené do *<select>* elementov. Všetky vhodné lokality sú preiterované a tie, ktoré boli v hodnotách z požiadavky, označím ako *selected*. Ak bola v požiadavke hodnota -1 alebo 0, znamená to, že bola vybraná hodnota All alebo, že nebolo vybrané nič, čo je pre algoritmus totožná informácia. To isté je vykonané aj s hotelmi, s jedným rozdielom, do pomocnej premennej - pre správne zobrazenie hodnôt v ďalších filtroch - je uložená informácia o tom, či bola zvolená hodnota All. Ak bola zvolená hodnota All, do filtra pre kategórie izieb, hviezdičky a samotné izby sú uložené hodnoty zo všetkých hotelov. Ak bol zvolený aspoň jeden hotel, tak sú do filtrov uložené len hodnoty prislúchajúce hotelu, resp. hotelom. Ak je teda zvolený hotel, ktorý má v ponuke len dvojposteľové izby, v kategórií izieb sa zobrazí len táto možnosť a pod. Taktiež sa v tomto prípade vyplnia aj konkrétne dáta do filtra pre samotné izby.

Následne sa pokračuje vo vyplňovaní HTML pre každý filter rovnako - hodnoty, ktoré prišli v požiadavke, sú označené ako *selected*, ostatné zostávajú ako klasické *<option>* elementy. Výsledné HTML je uložené v premennej, pre každý filter pod iným idexom v poli a vrátené na konci funkcie. V JS `success` časti ajax volania sú vyplnené dané *<select>* elementy hodnotami, ktoré sú z funkcie vrátené. Filtrácia v tomto momente funguje a to, čo potrebujem doprogramovať je samotný výpis zvolených hotelov.

Výpis sa vykoná po JS `onclick` funkcii na tlačidlo Filter. Podobne ako v dynamickej filtrácii sa zavolá ajax, avšak v tomto momente sú podstatné len zvolené hodnoty. Obdobne sú z databázy vybrané požadované hotely na základe typu užívateľa. Ak sú zvolené konkrétne hotely, sú vybrané tie, ak sú zvolené len lokality, sú vybrané všetky hotely v nich. Následne sa aplikuje filter hviezdičiek, kategórií a konkrétnych izieb. Na všetky filtrácie, resp. výbery sú používané Laravel vbudované funkcie, ktoré sa aplikujú nad kolekciami, ktoré vracajú model a fungujú rovnako ako SQL dotazy. Na kolekciu je možné použiť napr. funkciu `where`, kde sú parametre názvy stĺpcov a hľadané hodnoty a pod. Ďalej je preiterovaný každý hotel, ktorý po filtrácii zostal a vygeneruje sa pre neho HTML blade, kde je využitý filter dní a dátumu - zobrazí sa HTML tabuľka na vybraný dátum a vybrané dni v týždni.

Druhá časť tejto úlohy zahŕňala presun a prepis konfigurátora. Táto časť bola už jednoduchšia, mohla som použiť množstvo kódu, ktorý som napísala v prvej časti. Ako som už spomenula, pre konfigurátor bola vytvorená samostatná URL. Konfigurátor sa zobrazuje pre každý hotel samostatne, pretože sa vždy upravuje len jeden hotel a jeho izby a nemalo by zmysel mať konfigurátor pri

filtroch. Konfigurátor obsahuje taktiež filtre, ale je ich menej - vzhľadom na to, že už je dané, aký hotel chce užívateľ upravovať, stačí len filter kategórie izby, resp. samotného výberu izieb. Následne sú na výber dni v týždni a dátum od-do, podobne ako v globálnych filtroch.

Oproti pôvodnej funkcionalite konfigurátora došlo k niekoľkým zmenám. Najväčšia bola pridanie možnosti výberu viacerých rozmedzí dátumov, ktorých sa dané úpravy dotknú. Možnosť viacerých rozmedzí ovplyvnila hlavne výpis v HTML. Zvyšné filtre v konfigurátore boli v podstate rovnaké ako v predchádzajúcich častiach zadania, opäť na seba nadväzovali. Konfigurátor umožňuje nastaviť viacero atribútov. V nasledujúcom texte ich popíšem.

Available/Not available: pomocou dvoch checkboxov je možné zmeniť dostupnosť vybraných izieb. Ak nie je zaškrtnutý ani jeden, hodnota v databáze sa nemení.

Rooms: tento *<input>* element umožňuje zmeniť počet voľných izieb daného typu. Hodnota nemôže byť záporná. Ak nie je zadaná žiadna hodnota, hodnota v databáze sa nemení.

Minimum Stay: tento *<input>* element umožňuje zmeniť minimálny počet nocí, na ktoré si musí zákazník pobyt zarezervovať. Ak nie je zadaná žiadna hodnota, hodnota v databáze sa nemení.

Release: tento *<input>* sa momentálne nepoužíva, do budúcna s ním však klient má v pláne pracovať.

Veľké množstvo kódu som použila z predchádzajúcej časti zadania, bol len upravený na použitie pre konfigurátor, princíp však ostal rovnaký. Vybrané hodnoty v konfigurátore sa po kliknutí na tlačidlo spracujú pomocou ajax volania. V prvom rade sú aktualizované dáta pre vybrané izby. V prípade, že užívateľ vyberie len kategóriu, aktualizujú sa údaje pre všetky izby v tejto kategórii z daného hotela. Po aktualizácii sa upravené izby preiterujú a pre každú izbu sa vypíšu príslušné hodnoty do tabuľky. Výsledkom tohto volania je vygenerovaná, resp. pregenerovaná tabuľka v pravej časti obrazovky s aktuálnymi hodnotami v rozmedzí, ktoré bolo upravované.

4.1.3 Zhrnutie postupu a prípadných problémov

Počas implementácie som narazila na niekoľko problémov, ktoré sa však podarilo efektívne vyriešiť.

Prvým z nich bola rýchlosť filtrácie - zo začiatku trvalo dlho, kým sa filtre prispôsobili vybraným hodnotám v predchádzajúcich filtroch, čo spôsobovalo nezrovnalosti v zobrazení výsledkov. Uvedené bolo následkom nesprávnych dotazov do databázy a nevyužitia možností, ktoré poskytuje Laravel s jeho vbudovanými funkciami, napr. umožňuje použitie SQL funkcie `with`, ktorá je v mnohých prípadoch rýchlejšia ako funkcia `join`. Hlavnou výhodou `with` je taktiež následné usporiadanie dát v kolekcii, ktorá je obdržaná z databázy. Riešením problému bolo znížiť počet dotazov do databázy za použitia Laravel funkcií tak, aby som z nich dostala všetky potrebné dáta, ktoré boli následne prechádzané a ďalej triedené.

Druhým bola taktiež rýchlosť, no nie filtrácie, ale zobrazenia daného výsledku filtrácie v podobe HTML. Tu bola chyba v tom, že som sa dané HTML snažila rozdeliť do čo najmenších a najprehľadnejších častí a v následnom výpise ich iteráciami vždy pomocou funkcie `@include` pripojiť k

výslednému HTML. Pri väčších výsledkoch však bolo pripojení až príliš, čo web extrémne spomaľovalo. Toto malo našťastie jednoduché riešenie. Všetky menšie časti som spojila do jednej väčšej, čo znížilo počet volaní `@include` funkcie a tak zrýchlilo beh celej stránky.

Samotný postup a rozdelenie podúloh boli korektné, časové odhady som taktiež splnila.

4.2 Algoritmus počítania cien

4.2.1 Zadanie úlohy

Úloha pozostávala z naprogramovania algoritmu, ktorý korektne vypočíta cenu na vybraný dátum, s danými parametrami a taktiež upravenia možnosti nastavenia sezón, či zliav, ktoré ovplyvňujú samotný výpočet. Súčasťou zadania bolo taktiež navrhnúť a naimplementovať, ako bude systém reagovať v prípade, že užívateľ nevyberie žiadny dátum. Algoritmus počítania cien bola najťažšia úloha, ktorá ma počas tejto praxe stretla.

Zdanlivo jednoduché zadanie pozostávalo z navrhnutia optimálneho riešenia, následnej úpravy databázy a množstva hodín strávených prepisovaním a upravovaním kódu. Počas implementácie tejto úlohy som narazila na mnohé problémy, avšak nakoniec ma programátorsky výrazne posunul.

4.2.2 Návrh riešenia a jeho implementácia

Rovnako ako v predchádzajúcej úlohe aj tento algoritmus bol už naprogramovaný. Čo sa však týka funkčnosti, nefungoval zo žiadneho hľadiska. Ceny sa získavali z databázy, avšak v nej neboli aktualizované a preto nemohli fungovať pri rôznych parametroch zadaných užívateľom - algoritmus vždy vrátil jednu, zdanlivo najnižšiu cenu, bez ohľadu na to, aké parametre užívateľ zadal. Jedno využitie však predchádzajúce naprogramovanie algoritmu malo, jednoducho som našla všetky miesta v kóde, kde sa daný algoritmus používa.

Vzhľadom na typ úlohy bolo najdôležitejšie naplánovanie samotného riešenia a analýza potrebných dát. Funkcia obsahujúca algoritmus sa používala v podstate na každej stránke portálu, takže dôležitá bola hlavne rýchlosť a taktiež to, aby sa výsledok z nej dal čo najlepšie použiť už v existujúcom kóde. Z tohto dôvodu som si ako parameter vo funkcii nechala informáciu o tom, či návratová hodnota má byť reťazec ceny spolu s dodatočnými informáciami alebo len samotná cena - to umožnilo jednoduchšiu spätnú reimplementáciu do už existujúceho kódu.

V prvom rade som upravila nastavenia niektorých entít, ktoré ovplyvňujú výpočet. Týmito entitami sú sezóny a zľavy. Sezónam som pridala možnosť prekrývania sa a taktiež priradenie minimálneho, resp. maximálneho počtu nocí, na ktorý môže užívateľ zarezervovať pobyt. Zľavám som pridala rovnakú funkcionalitu a taktiež, tzv. "Discount actions", ktoré popíšem pri samotnom algoritme. V prípade minimálneho, resp.maximálneho počtu nocí, je pri zľavách, oproti sezónam, rozdiel. Jedna zľava môže mať niekoľko rozmedzí minimálneho - maximálneho počtu nocí a pre každú z nich,

môže byť aplikovaná iná percentuálna zľava. Tá správna sa vyberie práve na základe počtu nocí. Ďalej popíšem entity, ktoré v algoritme figurujú.

Hlavnou entitou je **hotel**, pre ktorý sa samotná cena počíta. Cenu ovplyvňujú **sezóny**, ktoré sú nastavené na dané obodobie. Ďalej cenu ovplyvňuje *typ stravy* počas pobytu - bez stravy, raňajky, polpenzia, plná penzia, all inclusive alebo ultra all inclusive. Každý typ má inú cenu pre každú izbu. Následne môže mať každý hotel **zľavy**, ktoré sú taktiež zadávané na isté obdobie. Zľavy sú rozdelené do troch hlavných kategórií - admin zľavy, valid_to zľavy a provízie. Všetky zľavy majú na výber rovnaké nastavenie v Systéme, rozdiel je v storno podmienkach. Tie sa líšia na základe toho, či sa zľava uplatňuje na tzv. Primary price alebo Secondary price - to zvolí super-admin, resp. admin alebo hotel manažér v Systéme. Na Secondary price sa väčšinou uplatňuje väčšia percentuálna zľava, avšak storno podmienky sú prísnejšie. V algoritme je, čo sa týka zliav, najdôležitejšie zistenie, či sa vôbec na daný termín majú započítať a ak áno, na ktoré dni z pobytu a následne ich správne započítanie podľa priority. Najväčšiu prioritu majú hotel zľavy, menšiu admin zľavy a potom provízie. Ďalej cenu ovplyvňujú tzv. **placement options**, ktoré obsahujú všetky kombinácie, ktoré môžu pri určitom druhu izby nastať z hľadiska obsadenosti osobami. V konkrétnej trojlôžkovej izbe môžu byť kombinácie, napr:

- jeden dospelý a dve deti do 6 rokov,
- jeden dospelý a dieťa od 7 do 17 rokov,
- dvaja dospelí a dieťa do 6 rokov,
- dvaja dospelí a dieťa od 7 do 17 rokov,
- traja dospelí,...

Kombinácií môže byť rôzny počet, to záleží na hoteli. Každá z týchto kombinácií má inú cenu. To slúži na vyfiltrovanie správnej kombinácie, ktorú užívateľ hľadá a tým pádom aj správnej ceny. Entita, ktorá cenu síce neovplyvňuje, avšak ovplyvňuje to, či sa má vôbec porovnávať, resp. zobraziť v algoritme je **allotment**, resp. obsadenosť izieb.

Pre správnu funkčnosť a hlavne zníženie počtu dotazov do databázy a tým zrýchlenie samotného algoritmu bolo nutné vytvoriť dve nové tabuľky. Tabuľka **room__real__prices** obsahuje cenu pre každú kombináciu jednej izby v kombinácii s každým typom stravy. Táto tabuľka sa aktualizuje pri upravení, resp. vytvorení novej izby a upravení sezóny, resp. jej cien. Na to, aby sa dalo zistiť, ktorá kombinácia má danú cenu, bola vytvorená druhá tabuľka **room__real__prices__combinations**, ktorá obsahuje pre jeden riadok z room_real_prices, príslušný počet riadkov, pre každú osobu v danej kombinácii, s odkazom do tabuľky, z ktorej zistím aké typy osôb sú v tejto kombinácii. Jednoduchý príklad:

- v tabuľke room_real_prices bude záznam s ID 1, s cenou pre izbu s ID 519 a jedlom s ID 10, pre dve dospelé osoby,

- v tabuľke room__real__prices__combinations budú dva záznamy pre ID 1, ktoré budú obsahovať ID 141,
- v tabuľke, kde sú uložené placement options bude záznam, kde bude ID 141 a pri ňom informácia, že je to osoba od 18 rokov do napr. 99 rokov.

Algoritmus pozostáva z dvoch implementačných častí, jednou je vypočítanie **najnižšej** ceny pre danú krajinu, lokalitu alebo hotel. Druhou je vypočítanie cien pre všetky izby v jednom hoteli. Popisovať budem postup v algoritme pre výpočet najnižšej ceny - ten obsahuje viac parametrov a môže v ňom nastať viac situácií. Vypočítanie cien pre všetky izby je už len zjednodušenou variantou tohto algoritmu. Čo sa týka najnižšej ceny algrotimus vráti práve jednu cenu a v prípade všetkých izieb všetky ceny vhodných izieb na základe zadaných parametrov. Parametre sú nasledovné:

- počet izieb a osôb v nich. V prípade, že sú v izbách aj deti, ich vek.
- ID entity, pre ktorú sa počíta najnižšia cena, resp. ceny izieb. Môže to byť krajina, lokalita alebo hotel.
- typ entity, pre ktorú sa počíta najnižšia cena.
- dátum, na ktorý užívateľ pobyt hľadá, resp. informácia o tom, že dátum nebol zadaný.

Tieto parametre sú všetko, čo potrebujem, zvyšok je možné zistiť pomocu nich. Následne krok po kroku popíšem funkcionalitu algoritmu.

V prvom rade zistím, či je zadaný dátum od užívateľa. To je jedna z najdôležitejších informácií a na základe nej sa algoritmus rozdeľuje na dve vetvy. To, či je dátum zadaný, sa zistí pomocou *Session*, do ktorej sa ukladá pri kliknutí na vyhľadávacie tlačidlo.

```
// check if there is date stored in Session
if (Session::has('searchstartdate') && Session::has('searchenddate')) {
    $startDay = Carbon::parse(Session::get('searchstartdate'));
    $endDay = Carbon::parse(Session::get('searchenddate'));
    $dateIsSet = true;
} else {
    $dateIsSet = false;
}
```

Listing 4.2: Ukážka kódu získavania informácií zo Session

Rovnakým spôsobom sa skontroluje to, či užívateľ nastavil vyhľadávané izby. To, že nebude v *Session* nastavená žiadna izba sa môže stať len v jednom prípade - pri prvej návšteve portálu. Následne sa *Session* vždy nastaví, ale pre tento prípad musí byť vytvorená aj *else* vetva. K izbám je

pridaná informácia o celkovom počte osôb, podľa ktorej sa ďalej vyhľadáva v tabuľke **room_real_-prices**. Následne sa z databázy vyberú hotely, v ktorých sa bude hľadať najnižšia cena. Tu zohráva úlohu to, či sa vyhľadáva najnižšia cena pre hotel, lokalitu alebo krajinu. Podľa toho sa vyberú aj dané hotely s využitím parametrov ID entity a typ entity. Ak bude entita hotel, z databázy je obdržaný jeden hotel, ak to bude napr. lokalita, sú obdržané všetky hotely v danej lokalite.

V tejto časti algoritmu dochádza k rozdeleniu na dve vetvy na základe toho, či je zadaný dátum alebo nie.

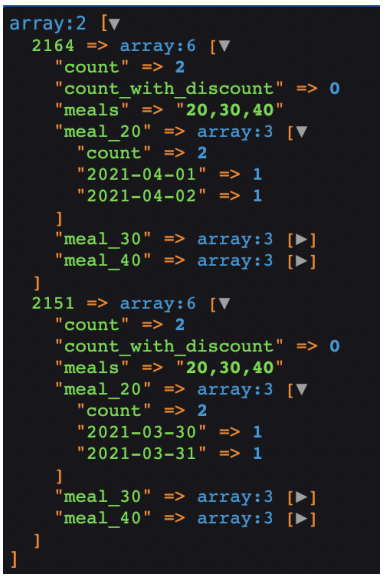
- ak je dátum zadaný, vyfiltrujú sa len tie sezóny v hoteli, resp. hoteloch, ktoré sú aktívne v daný dátum. Je nutné filtrovať aj sezóny, ktoré sú aktívne len **časť** daného obdobia a to z dôvodu, že pobyt môže byť na prelome sezón. S hotelmi sa vyfiltrujú v prvom dotaze všetky informácie, ktoré sú potrebné. Z môjho pohľadu je to najefektívnejšie a najčistejšie riešenie, čo sa týka kódu. Každý dotaz do databázy spomaľuje rýchlosť algoritmu a to určite nie je žiadúce.

- v prípade, že dátum nie je zadaný je dotaz jednoduchší. Vyhľadáva sa najnižšia cena na rok dopredu. Tento časový interval bol navrhnutý klientom.

V obidvoch prípadoch sú zatiaľ nepodstatné zľavy - do cien bez zadania dátumu sa nezarátavajú a pri zadaní dátumu sa vyhľadajú ďalej v kóde.

Ďalšia časť algoritmu je mierne odlišná v prípade, že je zadaný dátum a v prípade, že zadaný nie je - rozdiel je v tom, či sa počíta do ceny zľava. Začne jeden veľký cyklus, ktorý prechádza každý jeden hotel v danej lokalite, resp. krajine, popr. len jeden hotel, ak je užívateľ na jeho stránke. Tu sa skontroluje to, či naozaj každý jeden deň pobytu je aspoň jedna sezóna. Môže sa totiž stať, že z databázy obdržím hotel, ktorý má síce sezónu, ale platí len počas časti pobytu a žiadna iná sezóna na ňu nenadväzuje. Zavolá sa funkcia, ktorá skontroluje túto skutočnosť a vracia *true* alebo *false* na základe toho, či je na každý deň aspoň jedna sezóna. Ak vráti *false*, tak cyklus pokračuje ďalším hotelom. Následne sa skontroluje, či počet nocí pobytu spadá do sezóny, resp. sezón, ktoré zostali. Táto informácia sa nemôže kontrolovať v predchádzajúcom kroku, kvôli korektným chybovým hláškam pre užívateľa. Ak po tejto filtrácii nezostane žiadna sezóna vyriešia sa chybové hlášky, ktoré sa zobrazia užívateľovi.

Ak hotelu zostali sezóny pokračuje sa ďalej. Tentokrát sa zavolá funkcia, ktorá vráti pole s informáciou koľko dní z pobytu, je v ktorej sezóne. To sa musí skontrolovať pre každý typ stravy. V prípade, že by jedeň den pripadol na dve sezóny, prioritu má novšia sezóna.



V ukážke je vyhľadaný pobyt na prelome sezón - dva dni sú v sezóne s ID 2164 a dva dni v sezóne s ID 2151. Tieto informácie sa využijú ďalej v algoritme.

Obr. 4.1: Ukážka uloženia dát v poli

Po kontrole sezón je zistené, aké zľavy platia v dané dni. Preiteruje sa každá zľava hotela, ktorá spĺňa minimálny, resp. maximálny počet nocí a vyhovujúce zľavy sú uložené do poľa. Následne sa preiteruje každý deň pobytu a uložia sa zľavy pre dané dni. To je nutné z dôvodu, že každý deň pobytu môže mať iný počet zliav a teda aj celkovú cenu. Je nutné vedieť, ktorý deň má aké zľavy. Keď sú každému dňu priradené prislúchajúce zľavy, vyhodnotia sa ešte tzv. "Discount Actions". Tieto akcie sa nastavujú v Systéme a určujú, na ktoré dni sa zľava aplikuje. Môžu byť nasledovné:

- zľava sa aplikuje len na dni, na ktoré platí, napr. ak platí len v pondelok a pobyt je od pondelka do piatku, aplikuje sa len v pondelok,
- zľava sa aplikuje na všetky dni pobytu, aj keď platí len v niektoré z nich, napr. ak platí len v pondelok a pobyt je od pondelka do piatku, aplikuje sa na všetky dni pobytu,
- zľava sa aplikuje len ak platí počas všetkých dní pobytu, napr. ak platí len v pondelok a pobyt je od pondelka do piatku, neaplikuje sa na žiadny deň.

Na základe daných akcií sa doplnia, resp. zmažú zľavy z niektorých dní. Pole s dátami vyzerá nasledovne `[deň][admin_zľava/hotel_zľava/provízia] = daná zľava`.

Po získaní zliav už sú pre algoritmus získané všetky potrebné dáta a prechádza sa k samotnému počítaniu. Preiteruje sa každá sezóna hotela. Pre každú sezónu sa vyfiltrujú len správne izby, najskôr podľa počtu osôb, následne aj podľa placement options. Ak po týchto filtráciach nezostanú vhodné izby, pokračuje sa na ďalšiu sezónu. Ak zostanú, iteruje sa cez každú izbu a kontroluje sa jej obsadenosť, minimálny, resp. maximálny počet nocí a to, či je v danom dátume aktívna. V momente, kedy aspoň na jeden deň niektorá z týchto požiadaviek neplatí, cyklus pomocou *break* pokračuje

ďalšou izbou. V prípade, že izba splní všetky požiadavky, prichádza na rad aplikovanie zliav. Ak žiadne zľavy nie sú, je ďalší postup jednoduchý - cena za noc sa vynásobí počtom nocí v sezóne a uloží do poľa. V poli musia byť uložené všetky ceny - pred zľavami, po zľave aj netto cena. V tomto prípade sú všetky ceny rovnaké. Avšak ak zľavy sú, prechádzajú sa postupne po dňoch a svojich prioritách. Prvé sú admin zľavy, následne hotel zľavy a nakoniec provízie, na každý deň samostatne. Tu sa využije pole, ktoré drží informácie o počte dní z pobytu v sezóne. Po každom dni so zľavou sa hodnota pod indexom `count` v danej sezóne o jedno zníži. Ak sú preiterované všetky dni so zľavou a pod indexom `count` je hodnota 0, znamená to, že každý deň mal zľavu a ceny sú uložené do poľa. Ak je hodnota väčšia ako 0, základná cena v sezóne sa vynásobí zostávajúcim počtom dní. Následne je možné pokračovať ďalšou izbou, resp. sezónou. Ak je pobyt na prelome sezón, do výsledného poľa sa podľa indexov v ďalšej iterácii ceny pripočítavajú. Indexy musia rozdeľovať izby podľa poradia vyhľadávanej izby (ak užívateľ vyhľadáva dve izby, je nutné mať ich pod samostatným indexom), ID samotnej izby, stravy a následne každá cena musí mať svoj index - pred zľavou, po zľave aj netto. V tomto poli je na konci iterácie pre každú izbu cena za daný pobyt. Posledným krokom je nájsť v poli najnižšiu cenu. V prípade, že sa funkcia volá z lokality alebo krajiny, je výsledkom pole najnižších cien pre každý hotel. Z tohto poľa sa následne získa finálna najnižšia cena.

Nakoniec sa vezmú do úvahy informácie o tom, či má funkcia vrátiť reťazec aj s typom stravy a menou alebo len samotnú cenu. Podľa toho sa vráti z funkcie daná hodnota. Samozrejme, môže sa stať, že sa nenašli vhodné izby a na konci funkcie bude pole prázdne. V tom prípade sa vráti z funkcie príslušná chybová hláška.

4.2.3 Zhrnutie postupu a prípadných problémov

Pri implementácii tohto algoritmu som narazila na nejeden problém a niekedy aj na svoje vtedajšie hranice programátorských schopností. Avšak vďaka konzultáciám a spolupráci na návrhu riešenia so seniornými kolegami som bola schopná úlohu úspešne dokončiť.

Rýchlosť algoritmu bola od začiatku najväčším problémom. Je zrejmé, že algoritmus potrebuje nemalé množstvo informácií, ktoré som najprv získavala neefektívnym spôsobom - vo viacerých dotazoch, či priamo v cykle. To pri vyhľadávaní najnižšej ceny, napr. v lokalite, či krajine spôsobovalo veľké problémy. Nakoniec som našla optimálne riešenie, ktoré väčšinu potrebných údajov získalo už na začiatku, jedným dotazom, a ďalej sa s dátami len pracovalo.

Ďalším problémom bolo množstvo parametrov, ktoré vplývali na samotný výpočet a ich správne nadviazanie na seba. Najproblematickejším z nich bol určite výpočet na prelome sezón - bolo nutné zobrať do úvahy množstvo vedľajších parametrov. Najťažšie bolo určiť, či v obidvoch sezónach je rovnaká možnosť stravy a následné napojenie správnych cien za dané typy stravy.

Celkovo však túto úlohu hodnotím kladne, donútila ma zaradiť vyšší stupeň, čo sa týka mojich programátorských schopností a preverila všetky potrebné aspekty. Negatívne by som zhodnotila svoje časové odhady v tejto úlohe, zabrala mi omnoho viac času, ako som pôvodne predpokladala.

4.3 Nová Google mapa

4.3.1 Zadanie úlohy

Nasledujúca úloha bola zameraná hlavne na programovanie v JS, resp. pripravenie si dát v PHP, čo najoptimálnejšie pre následné spracovanie pomocou JS. Jednalo sa o rozsiahlu úpravu vzhľadu a taktiež informácií zobrazených na mape. Úprava vzhľadu zahŕňala nasledujúce:

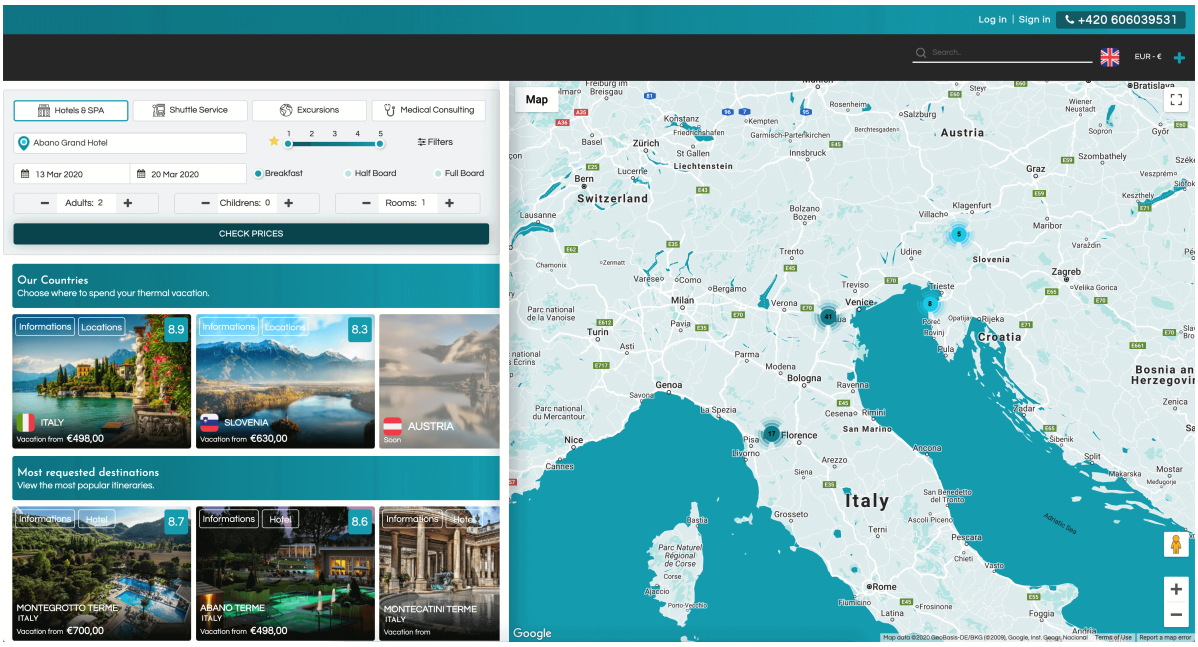
- zmenu celkového vzhľadu mapy,
- vytvorenie preklikov odkazujúcich na miesta na mape,
- optimálne prispôsobenie sa oddialenia mapy na základe veľkosti krajiny, resp. lokality,
- pridanie nových značiek s rôznymi atribútmi pre lokality a hotely,
- pridanie novej funkcionality spojenej so vstupmi od užívateľa,
- pridanie novej funkcionality vyskakujúceho okna.

Úloha nadväzovala na algoritmus počítania cien, pretože v novej mape sa mali taktiež zobrazovať informácie o najnižších cenách v daných lokalitách, resp. hoteloch v lokalitách.

4.3.2 Návrh riešenia a jeho implementácia

Predchádzajúca implementácia bola, čo sa efektivity týka, opäť na veľmi nízkej úrovni. Napríklad JS na zobrazenie mapy bol na každej jednej stránke, kde sa mapa nachádzala, namiesto toho, aby sa JS kód nachádzal v jednom súbore a len volal pomocou funkcie. Úprava práve tejto skutočnosti bola prvým krokom v rámci môjho riešenia.

Pre mapu som vytvorila nový JS súbor, kde som preniesla inicializáciu pôvodnej mapy a otestovala, či všetko funguje. Keď bolo všetko funkčné, prišiel rad na zmenu vzhľadu mapy. Tomu predchádzala analýza Google Maps API, čo v tejto úlohe zabralo najviac času. Dokumentácia od spoločnosti Google je veľmi detailná a pripravená pre programátorov aj s názornými ukážami samotnej inicializácie, či JSON súboru.



Obr. 4.2: Pôvodný vzhľad mapy

Ďalším krokom bola zmena z mapy farebnej, na mapu klasickú. Toto obnášalo zmenu JSON súboru v inicializačnej časti kódu. Pomocou dokumentácie som v JSON súbore prepísala štýly mapy - odstránila som farby, ktoré predchádzajúci programátori použili a nahradila ich predvoleným nastavením od Google, popr. farbami, ktoré požadoval klient. Každá časť mapy sa štýluje samostatne, takže bolo potrebné prejsť celý tento JSON. V rámci tejto časti som taktiež zmenila veľkosť maximálneho oddialenia mapy, ktoré sa nastavuje opäť v inicializačnej časti JSON súboru *minZoom: *hodnota**.

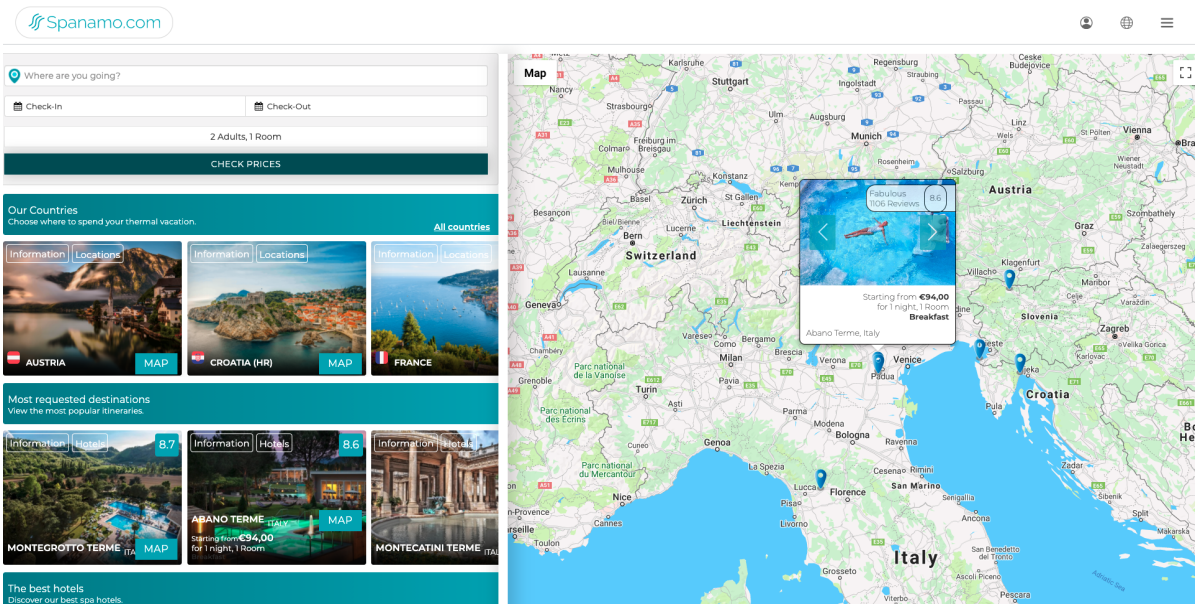
Následne som naprieč celým webom vytvorila prekliky na rôzne miesta na mape. Boli to prekliky na krajiny, lokality, či samotné hotely. V databáze na to boli pripravené dva stĺpce s informáciou o geografickej šírke, resp. dĺžke. Na základe týchto dvoch stĺpcov som na všetkých miestach, kde to klient požadoval, vytvorila tlačidlo, ktoré v *data* atribútoch obsahovalo tieto informácie a na `onclick` funkciu sa vykonal JS kód, ktorý mal za úlohu na dané miesto vycentrovať mapu. Tlačidlo má taktiež triedu, podľa ktorej je možné určiť, či sa jedná o krajinu, lokalitu alebo hotel. Na základe toho sa zvolí, na akú vzdialenosť sa má priblížiť mapa. API na tieto funkcionality obsahuje funkcie `setZoom` a `setCenter`. Do funkcie `setCenter` sa posielajú práve informácie o geometrickej šírke a dĺžke. Po dokončení tejto časti sme spolu s klientom zistili, že pri krajinách táto funkcionalita nefunguje optimálne. Mapa sa síce vycentrovala, ale priblíženie bolo pre každú krajinu rovnaké a to nebolo správne (pre lokalitu, či hotely to nebolo potrebné vzhľadom na ich veľkosť). Musela som teda nájsť riešenie. Analýzou tohto problému som strávila nemalé množstvo času a prešla som rozsiahlu dokumentáciu od Google Maps. Prišla som na tri riešenia, z ktorých som vybrala to najoptimálnejšie. Toto riešenie zahŕňalo úpravu databázy aj samotného kódu. Google API poskytuje

funkciu `fitBounds`, ktorá na základe zadaných súradníc primerane nastaví oddialenie mapy. Potrebuje však štyri súradnice, ktorými sú zemepisná šírka aj výška ľavého spodného rohu a pravého horného rohu obdĺžnika, ktorý ohraničuje danú krajinu. Do databázy som pridala tabuľku, ktorá všetky štyri súradnice uchováva a do krajiny pridala cudzí kľúč, ktorý na ne odkazuje. V Systéme som ku krajine pridala štyri nové *<inputy>*, do ktorých sa dané informácie vypĺňajú. V kóde to znamenalo pridať tieto štyri hodnoty do *data* atribútov a taktiež pridať podmienku, ak sú dané atribúty vyplnené, použiť funkciu `fitBoundries` a ak nie, použiť `setCenter`.

Po tejto náročnej časti nasledovalo pridanie úplne novej funkcionality pop-up okien, resp. Info Windows, ako sa nazývajú v dokumentácii od Google. Tieto okná obsahujú informácie o miestach, ktoré sú na mape vyznačené. K tomu, aby som sa dostala k samotným pop-up oknám, však musím najprv vysvetliť, čo sa na mape samotnej vyznačuje. Sú to buď lokality alebo hotely a to na základe toho, čo užívateľ vyhľadá. Napríklad, keď vyhľadá Taliansko, tak sa značky zobrazia len v Taliansku. Keď vyhľadá lokalitu, napr. Abano Terme, značky sa zobrazia len v nej. Čo sa týka hotelov, ak vyhľadá hotel, zobrazia sa všetky hotely v lokalite, do ktorej hotel spadá. Na portáli sú taktiež rôzne typy značiek a to pre lokalitu tmavomodré - dva typy, buď s červeným krúžkom v strede (v prípade, že žiadny hotel v lokalite nemá na daný dátum voľnú izbu) alebo s prázdnym krúžkom v strede (ak existuje aspoň jeden hotel, ktorý má voľnú izbu, ktorú užívateľ hľadá). To isté platí pre hotely, ktoré majú svetlomodré značky, až na jednu výnimku - hotel môže byť tzv. bestseller a v tom prípade má značku, ktorá obsahuje miesto prázdneho krúžku, krúžok s hviezdou. Informácia o tom, či je hotel bestseller, je uložená v databáze. Pred zobrazením mapy sa teda musia lokality, resp. hotely vytriediť. Najprv tie správne, v danej krajine alebo lokalite a následne na tie, ktoré majú voľné miesto a tie, ktoré voľné miesto nemajú. To sa deje v Controlleri, pomocou algoritmu cien z predchádzajúcej úlohy. Do algoritmu sa pošlú všetky hotely z lokality. Následne sa na základe výsledku z algoritmu vyplní pomocný atribút *isEmpty*, ktorý slúži na určenie toho, akú značku priradiť hotelu. Tak isto sa vyhodnotí, akú značku priradiť lokalite. Samotný pop-up je len HTML vyplnené dátami. Toto HTML som priradila ešte v Controlleri, pre každý hotel a každú lokalitu.

Celý proces sa vykonáva v jednej funkcii. Funkcia vracia pole, ktoré obsahuje všetky lokality aj hotely s potrebnými dátami. Následne toto pole zakóduje v JSON súbore a po načítaní stránky pošle do JS pre inicializáciu mapy. Počas inicializácie sa pre každý element v JSON objekte vykoná časť kódu jemu prislúchajúca. Najprv sa skontroluje, či je to lokalita alebo hotel. Následne podľa toho, do ktorej časti podmienky sa dostane, sa kontroluje, či je lokalita, resp. hotel voľná/ý. V prípade, že je to hotel aj to, či je to bestseller. Podľa toho sa priradí správna značka. Potom sa ešte na každú značku pridá *eventListener*, ktorý na `mouseover` zobrazí pop-up a zmení značku na svetlejšiu. Na `mouseout` zmení značku naspäť na tmavšiu, ale pop-up nezatvára. Každej značke sa taktiež priradí URL, na ktorú má po kliknutí presmerovať. V našom prípade je to stránka danej lokality alebo hotela.

Výsledná mapa spolu s pop-up oknami vyzerá takto:



Obr. 4.3: Nový vzhľad mapy

4.3.3 Zhrnutie postupu a prípadných problémov

Táto úloha bola v podstate bezproblémová. Jediným problémom, ktorý nastal bolo spomenuté korektné oddialenie a vycentrovanie mapy na základe veľkosti krajiny. To som však vyriešila preskúmaním dokumentácie Google Maps API.

Čo sa týka časovej náročnosti zadania, podúlohy som plnila v odhadnutom čase, avšak analýza a samotné preštudovanie dokumentácie mi trvalo dlhšie ako som predpokladala. Po dôkladnom prečítaní dokumentácie som však bola schopná rýchlejšie a efektívnejšie navrhovať riešenia týkajúce sa mapy.

4.4 Nový grafický design a logika stránok

4.4.1 Zadanie úlohy

Poslednou úlohou bola prevažne zmena grafického designu stránok. Predchádzajúci vzhľad portálu bol v rovnakom stave ako samotný kód - z veľkej časti chybný. Responzívna časť v podstate nefungovala a desktopový design síce áno, avšak samotné štýlovanie v SCSS súboroch bolo neprehľadné.

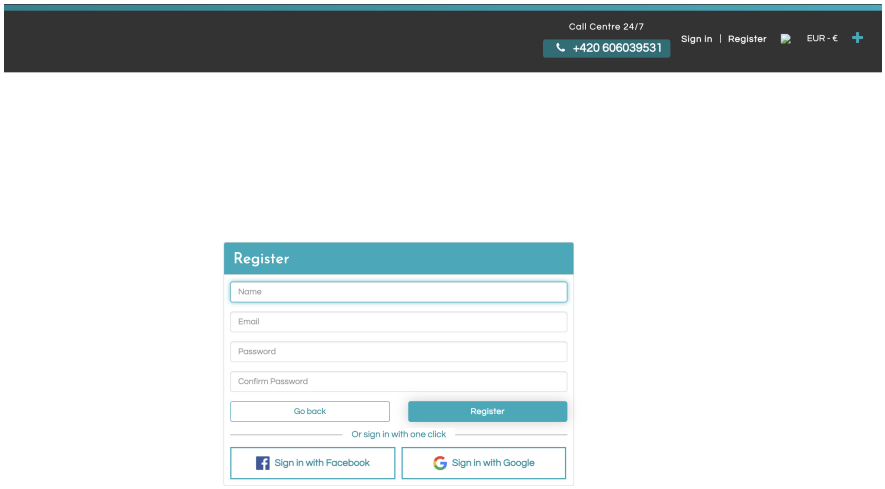
Nový grafický design si vytváral klient sám, no často ho so mnou konzultoval, poprípade ho upravoval po nasadení na testovací server. Východzí návrh bol vždy na desktop o šírke 1440px. V tomto rozmere musel byť vždy tzv. "pixel perfect", a teda na pixel presný. Následne som vždy dostala jeden návrh na mobilné zariadenie, ktorý bol pre mňa šablónou na všetky mobilné, resp.

tabletové zariadenia. Nebolo presne zadané, v ktorom rozmere sa musí desktopový design meniť na mobilný a v tejto časti som mala voľnú ruku.

Aj keď zadaním bol len nový grafický design, vyžadovalo to nielen vytvorenie HTML šablón a SCSS kódu pre štýl, ale taktiež zásah do PHP logiky, či vytvorenie JS skriptov.

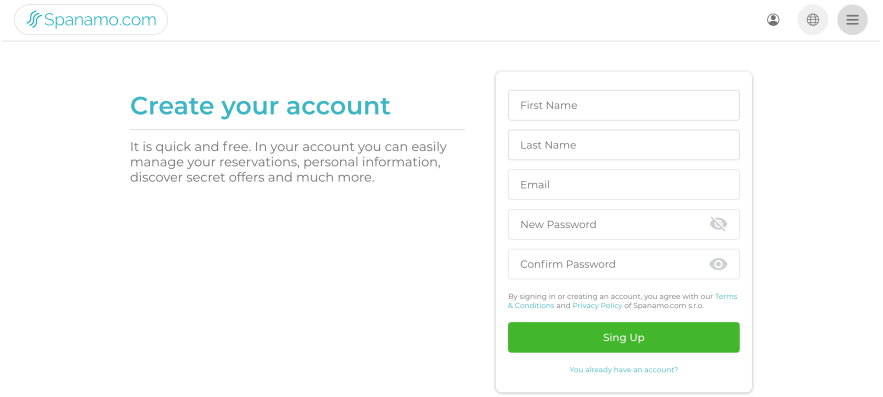
4.4.2 Návrh riešenia a jeho implementácia

Samotným redesignom prešiel celý web, v tejto bakalárskej práci však detailne popíšem len niektoré stránky. Pôjde o autorizačné stránky - prihlásenie a registrácia, taktiež o stránku, ktorá slúži na kontrolu rezervácie pre nezaregistrovaných užívateľov, užívateľský profil s možnosťou zmeny dát a v neposlednom rade hlavičku, pätičku a bočné menu. Mojím cieľom bolo naprogramovať stránky tak, aby ich štýlovanie bolo jednotné.



Obr. 4.4: Pôvodný vzhľad stránky pre registráciu

V priloženom obrázku je zobrazený pôvodný grafický design portálu, resp. jeho registračnej stránky a hlavičky. Prihlasovacia stránka vyzerala obdobne a stránka kontroly rezervácie či pätička neexistovali. Ako môžeme vidieť, HTML šablóna je naprogramovaná nevhodne, registračný formulár nie je zarovnaný zo žiadnej strany. Tak isto hlavička - už na prvý pohľad pôsobí na užívateľa nepríjemne, obsah nie je horizontálne zarovnaný na stred navigačnej lišty. Nový grafický design naopak pôsobí na užívateľa príjemne, jemne a moderne. Neobsahuje zbytočne veľa rušivých elementov a má jednotný design. Drží sa hlavných farieb portálu, ktorými sú modrá, šedá a biela, avšak v iných odtieňoch, ako v pôvodnom designe.



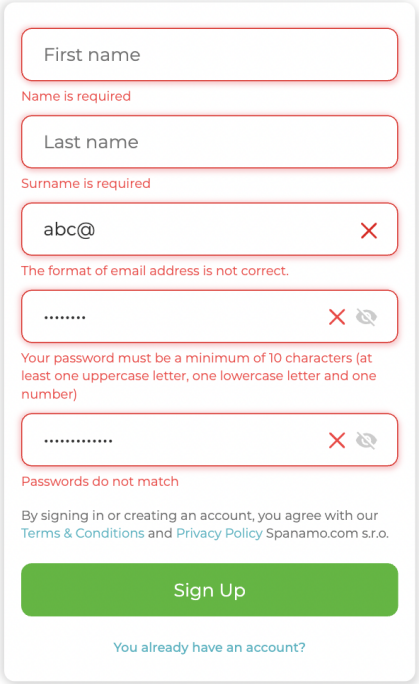
Obr. 4.5: Nový vzhľad stránky pre registráciu

Ďalej popíšem postup pri riešení zadanej úlohy. V prvom rade som zmazala celý HTML a SCSS kód, ktorý naprogramovali predchádzajúci vývojari. Kód som nechcela upravovať, ale napísať úplne nanovo a to po všetkých stránkach - HTML, JS, SCSS aj prepojenie na PHP.

Následne som si detailne prezrela všetky stránky a premyslela si ich implementáciu. V projekte používame knižnicu bootstrap, ktorá má výborné využitie pri tvorbe responzívnych stránok a tak som vždy hľadala čo najlepšie rozloženie daných stránok do riadkov a stĺpcov, s ktorými bootstrap pracuje. Stránky prihlásenia, registrácie a kontroly rezervácie pre neregistrovaných užívateľov mali všetky totožné rozloženie a tak som začala s nimi.

Vytvorila som pre ne rovnakú štruktúru v ich PHP šablónach a HTML elementom rovnaké triedy, pre ich zacielenie pomocou SCSS. Stránka bola rozložená do dvoch stĺpcov, kde bol vždy v jednom nadpis a informačný text a v druhom formulár na vyplnenie príslušných dát. Prvý krok vždy obsahoval len desktop verziu návrhu. Celý návrh bol na bielom pozadí a obsahoval len jednoduché *<hr>* elementy na oddelenie obsahu. Na odlíšenie formulára v pravom stĺpci od pozadia som použila CSS atribút *bow-shadow*, ktorý vytvoril tieň celému elementu, a tým ho oddelil od pozadia. Ďalej som musela vyriešiť chybové vstupy od užívateľov. Tie mohli byť nasledujúce:

- užívateľ nezadá meno alebo priezvisko,
- užívateľ sa snaží zaregistrovať pod emailom, ktorý už existuje,
- užívateľ zadá email v zlom formáte,
- užívateľ zadá krátke heslo - heslo musí obsahovať aspoň 10 znakov,
- užívateľ zadá slabé heslo - heslo musí obsahovať aspoň 1 malé písmeno, 1 veľké písmeno a 1 číslo,
- potvrdzovacie heslo sa nezhoduje s prvým zadaným heslom od užívateľa.



Obr. 4.6: Nový vzhľad portálu - chybové vstupy

Chybové vstupy som musela vyriešiť nielen po stránke SCSS, a teda zobraziť užívateľovi červenú chybovú hlášku a zmeniť farbu *<input>* elementu, ale taktiež pomocou JS zavolať ajax požiadavku, ktorá skontrolovala dané vstupy. Toto všetko dokáže síce Laravel sám - má na to vytvorené svoje URL, na ktorých všetko skontroluje, avšak chybové hlášky a ich vzhľad má taktiež svoje, čo nebolo v súlade s naším grafickým designom a tak som autentifikáciu musela vytvoriť sama. To však bolo pomerne jednoduché, Laravel obsahuje triedu `Validator`. Pomocou tejto triedy sa dajú nastaviť pravidlá, podľa ktorých sa vyhodnotí, či sú dané vstupy správne alebo nie.

Trieda obsahuje funkciu `fails`, ktorá vracia hodnotu *true* v prípade, že v kontrolovaných hodnotách je chyba. Taktiež obsahuje funkcie `errors` alebo `messages`, ktoré zobrazia, ktorá konkrétna chyba nastala.

```
public function checkRegisterData(Request $request) {
    // check if email is unique and passwords match
    $validator = Validator::make($request->all(), [
        'email' => 'required|max:255|unique:users',
        'password' => 'required|regex:/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{10,}+$/',
        'password_confirm' => 'required|same:password'
    ]);

    $data = [];
    // if validator fails - error occurs
    if($validator->fails()) {
        $data['error'] = true;
        return json_encode($data);
    } else {
        $user = User::create($data); // create user
        // if user made reservations before they will be added to their account
```

```
        BookingMain::where('email', $request->email)->whereNull('user_id')->update
            (['user_id' => $user->id]);
        Bouncer::assign('client')->to($user); //assign role to user
        auth()->login($user); // login user
        return json_encode(["emailExists" => false]);
    }
}
```

Listing 4.3: Ukážka kódu registrácie

V prípade, že nastala chyba, sa na základe typu chyby zobrazí daná chybová hláška. Táto hláška už je vypísaná v HTML, avšak má CSS atribút *display* nastavený na hodnote *none*. Po kontrole sa zobrazí korektná hláška, ktorú vyberiem na základe zvolených data atribútov pomocou JS. Daným elementom sa nastaví atribút *display* na hodnotu block a elementom, ktoré je potrebné graficky upraviť, sa pridá dynamicky trieda *error*. Zvyšok už je naprogramovaný v SCSS. HTML elementom, ktoré obsahujú text, som priradila triedu "red", ktorej som v súbore *variables.scss* nastavila červenú farbu, a teda všetky elementy s touto triedou sú vo východzom stave červené. Súbor *variables.scss* slúži na nastavenie premenných, ktoré je následne možné používať v ostatných SCSS súboroch.

```
input {
  width: 100%;
  padding: 12px 20px 12px 20px;
  border-radius: 10px;

  &.error {
    box-shadow: $redBoxShadow;
    border-color: $redError;
    color: $redError;
  }
}

.email {
  &-error {
    svg {
      fill: $redError;
    }
  }

  &-ok {
    svg {
```

```
      fill: $successGreen;
    }
  }
}
```

Listing 4.4: Zjednodušená ukážka SCSS kódu pri chybovom vstupe

V ukážke vidíme výhody SCSS oproti CSS a to zanorovanie, či využívanie spomínaných premenných.

Až po úplnom dokončení desktopovej časti a schválení klientom som pokračovala v programovaní mobilnej verzie. V tomto bode je najdôležitejšie korektné rozloženie elementov, ktoré bolo potrebné zohľadniť už pri programovaní desktopovej verzie. Vzhľadom na to, že som na daný fakt vopred myslela, spolu s použitím bootstrapu a jednoduchým dizajnom bolo programovanie responzívnej časti webu rýchle a nevyžadovalo veľké zásahy do desktopovej verzie. Vo väčšine prípadov sa obsah ukladá pod seba, aby mohol byť čo najväčší a užívateľovi sa s ním dalo príjemne pracovať aj na menších zariadeniach. Týmto smerom sa vydal aj klient, obsah poskladal pod seba. Rozhodnutie, kedy sa bude meniť desktopové zobrazenie na to mobilné, nebolo klientom špecifikované a tak som sa snažila to čo najviac spojiť s bodmi zlomu v bootstrap knižnici. Po dokončení týchto troch stránok som k nim doprogramovala novú pätičku, hlavičku, ktorá obsahuje viacero podmenu a taktiež bočné menu, ktoré má informatívny charakter.

Nová hlavička, na rozdiel od tej predchádzajúcej, obsahuje len najpodstatnejšie informácie, je prehľadná a príjemná pre užívateľa. Rozloženie obsahu je nasledujúce: na ľavej strane je logo portálu a na pravej strane tri rozklikávacie ikonky. Prvá z nich odkazuje na autorizačné stránky a stránku na kontrolu rezervácie, resp. ak je užívateľ prihlásený, odkazuje na hlavné časti jeho profilu - osobné informácie, obľúbené hotely a zoznam rezervácií - a odhlásenie. Druhá umožňuje zmeniť jazyk, v ktorom sa zobrazuje obsah portálu a tretia zobrazuje bočné menu. Vzhľadom na to, ako je obsah hlavičky rozdelený, bolo jednoznačne najlepšou voľbou na štýlovanie zobrazenie s hodnotou *flex* a následné rozloženie *space-between*, ktoré všetky podelementy rodiča rozdelí tak, aby mali medzi sebou rovnaké medzery s tým, že bočné elementy ostanú úplne na kraji.

Všetky podmenu sa zobrazujú/skrývajú na JS funkciu `onclick` na danú ikonku. JS kód som sa snažila naprogramovať čo najvšeobecnejšie, aby som pre každú ikonku nemusela robiť *if*, *else* vetvu. To som dosiahla tak, že elementy, na ktoré bola naviazaná `onclick` funkcia, obsahujú *data* atribút, ktorý hovorí, aké podmenu má daná ikonka zobraziť, resp. skryť. Nasledujúca skrátená ukážka kódu priblíži túto funkcionalitu.

```
<li data-show="accountSubNav" class="nav-list-item login"> <!-- ACCOUNT -->
  <div class="user-subnav main-subnav">
    {!! getFlaticon('account-nav', 1) !!}
  </div>
  <div class="nav-subnav-wrapper" data-name="accountSubNav">
    <!-- content -->
  </div>
</li>
<li class="nav-list-item" data-show="languagesSubNav"> <!-- LANGUAGES -->
 <div class="main-subnav">
   {!! getFlaticon('global-nav', 1) !!}
 </div>
 <div class="nav-subnav-wrapper" data-name="languagesSubNav">
   <!-- content -->
 </div>
</li>
```

Listing 4.5: Zjednodušená ukážka HTML kódu časti hlavičky

V ukážke je vidieť, že *<li>* element obsahuje *data-show* atribút, ktorého hodnota je rovnaká ako hodnota *data-name* atribútu * * elementu, ktorý obsahuje podmenu. V JS následne stačí, keď sa skontroluje, čo obsahuje *data-show* kliknutého elementu a zobrazí, resp. skryje sa element, ktorý má tú istú hodnotu v *data-name*.

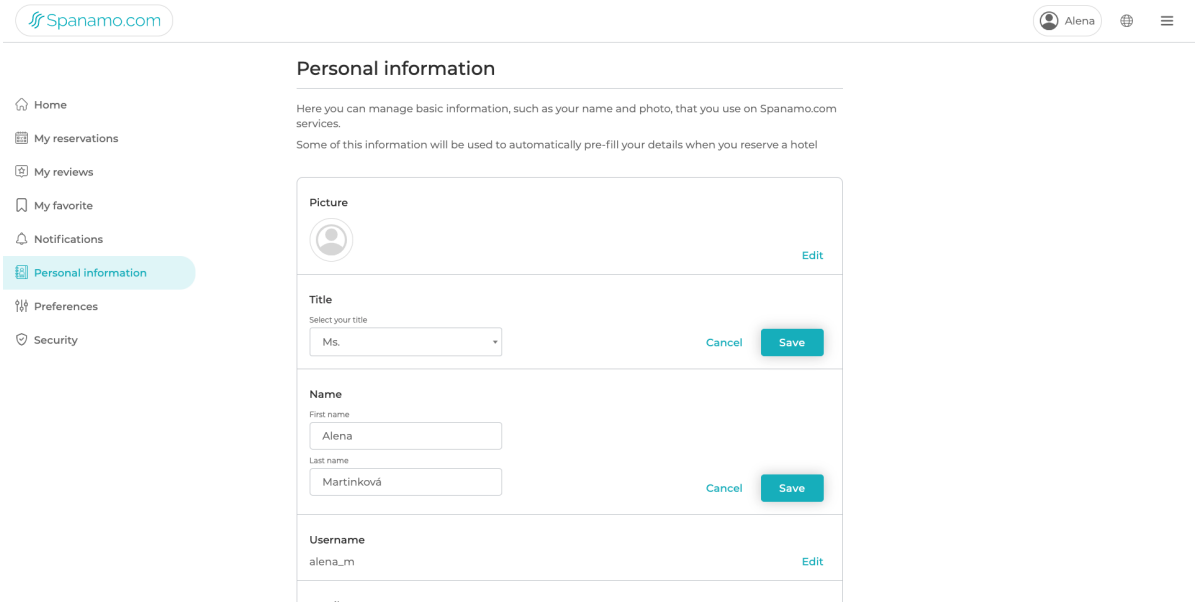
Všetky podmenu sa zobrazujú hneď pod ikonkou - sú napozicované absolútne, tak aby boli zarovnané sprava, presne podľa ikonky. Iba bočné podmenu, ako názov naznačuje, vychádza z boku obrazovky a je na celú výšku obrazovky. Zvyšok obrazovky, ktorú na šírku nezakryje, prekryje element, ktorý obsahuje čierne pozadie s atribútom *opacity*. V responzívnej verzii sa všetky podmenu zobrazujú na celú obrazovku, zmenila som len veľkosti fontu a niektorých elementov, popr. niektoré odsadenia.

Poslednou časťou, ktorej sa v tejto podkapitole budem venovať, je užívateľský profil s možnosťou úprav informácií. Užívateľský profil obsahuje 8 podstránok: Home, My reservations, My reviews, My favorite, Notifications, Personal information, Preferences a Security. V krátkosti popíšem, čo každá z nich obsahuje, resp. na čo slúži.

- **Home** - obsahuje textovú informáciu o tom, čo všetko si užívateľ môže zmeniť, resp. čo všetko sa nachádza v časti jeho profilu
- **My reservations** - zoznam užívateľových rezervácií, rozdelené sú do dvoch častí - budúce, resp. prebiehajúce a ukončené, resp. zrušené

- **My reviews** - zoznam užívateľových recenzií
- **My favorite** - zoznam užívateľových obľúbených hotelov
- **Notifications** - upozornenia pre užívateľa
- **Personal information** - osobné informácie o užívateľovi s možnosťou editácie
- **Preferences** - nastavenie jazyka a meny
- **Security** - zmena hesla a zrušenie účtu

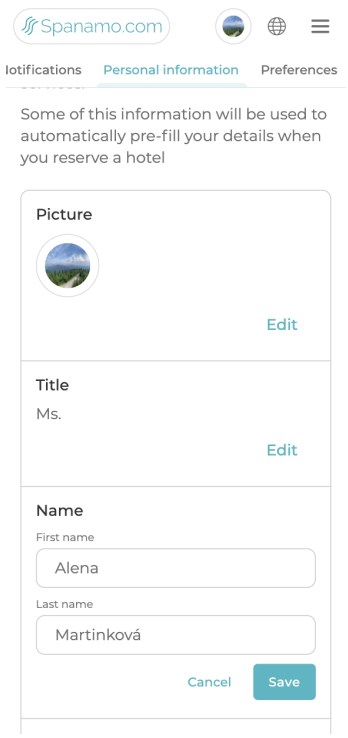
Každá stránka má rovnaké rozloženie obsahu, čo je vhodné ako na naprogramovanie, tak aj pre užívateľa a jeho vnímanie portálu. V ľavej časti sa nachádza menu so všetkými spomenutými podstránkami, aktívna stránka je vždy vyznačená modrou farbou. V pravej časti je obsah, ktorý je rozložený do riadkov. Vzhľadom na toto rozloženie som sa skoro vo všetkých stránkach vydala cestou *<table>* elementu. V prípade stránok My reservations a My favorites som sa rozhodla len pre * * elementy, pretože obsahovali viacero informácií a štýlovanie v tabuľke by nebolo efektívne. Vzhľadom na to, že stránky majú totožné rozloženie, detailnejšie popíšem len stránku Personal information, pretože obsahuje ako HTML a SCSS, tak aj JS a PHP funkcie pre dynamické zmeny dát v databáze.



Obr. 4.7: Časť stránky Personal information

V ľavej časti je menu so všetkými stránkami a v pravej časti obsah. Obsah je vždy rozložený nasledovne: nadpis oddelený čiarou, pod čiarou informačná veta a následne tabuľka s údajmi. Každý riadok obsahuje príslušné údaje, ktoré môžu byť upravené a text Edit. Údaje, ktoré môžu byť upravené sú: užívateľský obrázok, oslovenie, meno a priezvisko, užívateľskú prezývku, email, telefónne

číslo, dátum narodenia, pohlavie, národnosť a adresu. Pri každej položke platí nasledovné: ak má užívateľ zvolenú hodnotu, zobrazí sa daná hodnota, ak nie, zobrazí sa veta, že si danú položku môže vyplniť. V prípade obrázku sa zobrazí predvolený obrázok portálu. Každá položka sa edituje samostatne a všetko funguje na báze JS a ajax požiadavky, ktorá volá PHP funkciu, ktorá vyhodnotí korektnosť údajov a uloží ich do databázy, prípadne vypíše chybovú hlášku. Text Edit vždy zmení ľavú časť tabuľky z textu na *<input>*, resp. *<select>* element, podľa toho, čo daný užívateľ upravuje a pravú časť na Cancel a Save. Tento JS funguje obdobne ako JS v hlavičke - text Edit vždy obsahuje *data* atribút, podľa ktorého vie, ktorý riadok v tabuľke má zmeniť. Čo sa týka kontroly na strane PHP, v prípade *<select>* hodnôt užívateľ nemôže vybrať nesprávne, takže to je pomerne jednoduché, stačí uložiť hodnotu do databázy a následne ju dynamicky vypísať, aby ju užívateľ videl hneď a nie až po obnovení stránky. V prípade *<input>* hodnôt to už je náročnejšie - v niektorých položkách je nutné kontrolovať správnosť. Meno a priezvisko či adresa kontrolu nevyžadujú, resp. je pomerne nemožné vyhodnotiť, čo sa považuje za správne a čo nie. Avšak, napr. email a užívateľská prezývka sú dva údaje, ktoré vyžadujú kontrolu unikátnosti alebo správnosti. Pri emaile je kontrola rovnaká ako pri registrácii a to, či je email v správnej forme a či už rovnaký neexistuje v databáze. Čo sa týka užívateľskej prezývky, kontroluje sa unikátnosť a to z dôvodu, že som pri implementácii prihlasovania pridala možnosť prihlasovať sa práve pomocou užívateľskej prezývky. Kontrola funguje rovnako ako pri prihlasovaní a registrácii, pomocou Laravel `Validator` triedy.



Obr. 4.8: Responzívna časť stránky Personal information

Responzívna časť funguje úplne rovnako ako desktopová. Obsahuje taktiež všetky informácie a texty. Mení sa len jej rozloženie. Najväčším rozdielom je umiestnenie bočného menu, ktoré je pod 992px - bod zlomu v bootstrap knižnici - umiestnené nad celý obsah a napojené na hlavičku. Sú z neho odstránené ikonky a zmenené označenie aktívnej stránky. Na rozdiel od desktopovej verzie sú položkoy v menu uložené vedľa seba a nie pod sebou. Aktívna stránka má pod názvom stránky *after* element naštýlovaný ako obdĺžnik s atribútom *border-radius* a napozicovaný absolútne tak, aby z neho bolo vidieť len polovicu. Z hlavičky je odstránené spodné ohraničenie a tým je spojená s menu. Ohraničenie je až pod samotným menu. To sa musí diať len v užívateľskom profile, čo zabezpečí premenná nastavovaná v Controlleri, ktorý dané stránky vykresľuje. Následne som pridala funkcionalitu, ktorá pomocou JS vycentruje aktívnu stránku v menu do jeho stredu.

Obsah, ktorý je v desktopovej časti na pravej strane, je v responzívnej časti rozložený na celú stránku. Pod 576px sa rozloženie v tabuľke zmení nasledovne: jeden riadok bude obsahovať *<th>* a **pod ním** *<td>*. V desktopovej verzii sú tieto elementy vedľa seba. To zaručí, že tlačidlá Edit, resp. Cancel a Save budú v riadku pod informáciami o užívateľovi. Tým pádom môže mať *<th>* aj *<td>* element celú šírku riadku a zostane dostatok miesta pre všetky informácie.

4.4.3 Zhrnutie postupu a prípadných problémov

Problémy, na ktoré som narazila v tejto úlohe neboli závažné.

Prvým z nich bolo nedodržanie pixel perfect dizajnu, ktorý je minimálne v jednom rozlíšení nutný. Po konzultácii s klientom som si však na túto dôležitú časť dala pozor a ďalej s tým už problémy neboli.

Druhým bolo logické prepísanie funkcionality autorizačných stránok. Ako už bolo uvedené, Laravel je schopný autorizácie sám, a tak som musela prísť na to ako obísť riešenie od Laravelu a zároveň byť čo najefektívnejšia. Toto by som nenazvala problémom ako takým, skôr výzvou.

Posledným problémom bolo štýlovanie pre Safari a Apple zariadenia. Na rozdiel od Chromu, popr. Android zariadení, Safari množstvo štýlovaní nepovoľuje a tak som musela prísť na to, ako optimálne nakódovať stránky na všetky zariadenia. To sa týkalo hlavne kódovania responzívnej časti stránky Personal information, ktorá má v sebe tabuľku - štýlovanie tabuľky v Safari ignoruje napr. *margin*, ktorý je v iných prehliadačoch, resp. zariadeniach použiteľný.

Vo všeobecnosti však svoj postup a rozdelenie do jednotlivých častí považujem za korektné, hlavne čo sa týka prvotnej implementácie desktopovej verzie, keď som až po úplnom dokončení začala implementovať verziu mobilnú. Čo sa týka časovej náročnosti a svojich odhadov, na mobilnú verziu som si pridelila nakoniec viac času, než bolo potrebné. To však bolo zásluhou toho, že desktopovú časť som si naprogramovala tak, aby mi responzívna zabrala čo najmenej času.

4.5 Časová náročnosť jednotlivých úloh

V tejto podkapitole zhrniem čas strávený na jednotlivých úlohách.

Úloha	Čas v dňoch
Filter/konfigurátor obsadenosti izieb hotelov	7
Algoritmus počítania cien	20
Nová Google mapa	10
Nový grafický design a logika stránok	18

Tabuľka 4.1: Čas strávený na jednotlivých úlohách

Kapitola 5

Záver

5.1 Uplatnenie znalostí dosiahnutých štúdiom

Počas práce na projekte som bola otestovaná z viacerých hľadísk. Modifikovala a vytvárala som funkcie pre backend a tak isto využívala naprogramované funkcie na frontende. Štúdium na vysokej škole mi určite pomohlo, predovšetkým v začiatkoch praxe. V prvom rade mi pevný základ pre programovanie poskytli predmety *Programovanie 1 a 2* a taktiež *Algoritmy 1 a 2*. Z ďalších predmetov by som učite vyzdvihla *Úvod do databázových systémov* a *Databázové a informačné systémy*, vďaka ktorým som bola schopná od začiatku pracovať s databázou bez väčších problémov. Nepochybne mi pomohol aj predmet *Vývoj informačných systémov*, ktorý ma uviedol do problematiky návrhových vzorov a ich výhod, či nevýhod. Pri úprave frontendu a programovaní novej grafiky som využila znalosti z predmetu *Vývoj internetových aplikácií* a taktiež *Užívateľské rozhrania*, vďaka ktorým som mohla klientovi poskytnúť zmyslupný feedback na jeho grafické návrhy.

5.2 Znalosti chýbajúce v priebehu praxe

Medzi chýbajúce znalosti by som určite zaradila efektívnu implementáciu riešení, ktorá mi spočiatku robila problémy. To však nebol problém, pretože návrhy riešenia so mnou vytvorili a skonzultovali seniorskí skúsenejší kolegovia. Tým som získavala potrebné skúsenosti, aby som niektoré ďalšie riešenia bola schopná spracovať a navrhnúť sama. Zo začiatku mi taktiež chýbal správny odhad, aký dlhý čas bude potrebný na spracovanie danej úlohy a vôbec som neprihliadala na čas potrebný na testovanie.

5.3 Novonadobudnuté znalosti na praxi

Čo sa týka programovacích znalostí, zdokonalila som sa v jazykoch PHP, JS a taktiež v práci s databázami. Prehĺbila som svoje znalosti aj z hľadiska návrhových vzorov, minimálne MVC, na

ktorom je celý projekt postavený. Taktiež som sa naučila pracovať s Gitom, ktorý považujem za potrebný a skvelý nástroj pre programátorov.

Naučila som sa pracovať v kolektíve a komunikovať s klientom aj s nadriadenými. V neposlednom rade som zistila aj to, aké dôležité je dobre komentovať kód a písať správnu dokumentáciu, čo som využila hlavne v úlohách, ktoré na seba nadväzovali, no pracovala som na nich s väčším časovým odstupom.

V nadväznosti na predchádzajúcu podkapitolu môžem vyzdvihnúť fakt, že som sa naučila správne si odhadnúť čas, ktorý bude potrebný na spracovanie danej úlohy spolu so všetkými testovacími aj administratívnymi časťami.

Taktiež som sa po čase naučila rozdeľovať si veľké úlohy na menšie podúlohy, ktoré zrýchlili a sprehľadnili celú implementáciu. S tým mi pomohla aplikácia Jira, ktorú sme používali. Z toho vyplynulo aj spomínané zlepšenie v odhadovaní času.

5.4 Dosiahnuté výsledky a ich celkové zhodnotenie

Absolvovanie bakalárskej praxe hodnotím zo všetkých hľadísk veľmi pozitívne a bola pre mňa veľkým prínosom. Nadobudla som nové skúsenosti, ktoré určite využijem vo svojom budúcom profesijnom živote.

Literatúra

1. *Documentation for PHP* [online] [cit. 2021-01-21]. Dostupné z: `https://www.php.net/docs.php`.
2. *Documentation for Laravel* [online] [cit. 2021-01-21]. Dostupné z: `https://laravel.com/docs/`.
3. *Documentation for HTML* [online] [cit. 2021-01-21]. Dostupné z: `https://html.com/`.
4. *Documentation for JavaScript* [online] [cit. 2021-01-31]. Dostupné z: `https://devdocs.io/javascript/`.
5. *Documentation for Docker* [online] [cit. 2021-03-05]. Dostupné z: `https://docs.docker.com/`.